More on Lambda Calculus

Reduction strategies

- Normal-order reduction: choose the left-most, **outer-most** redex first

  (λu. λv. v) ((λx. x x)(λx. x x))
  → λv. v

  Normal-order reduction will find normal form if exists

- Applicative-order reduction: choose the left-most, **inner-most** redex first

  (λu. λv. v) ((λx. x x)(λx. x x))
  → (λu. λv. v) ((λx. x x)(λx. x x))
  → ...

Non-terminating reduction

(λx. x x) (λx. x x)
→ (λx. x x) (λx. x x)
→ ...

(λx. x x y) (λx. x x y)
→ (λx. x x y) (λx. x x y) y
→ ...

(λx. f (x x)) (λx. f (x x))
→ f ((λx. f (x x)) (λx. f (x x)))
→ ...

Reduction strategies – examples

Normal-order	***Applicative-order***
(λx. x x) ((λy. y) (λz. z))	(λx. x x) ((λy. y) (λz. z))
→ ((λy. y) (λz. z)) ((λy. y) (λz. z))	→ (λx. x x) (λz. z)
→ (λz. z) ((λy. y) (λz. z))	→ (λz. z) (λz. z)
→ (λy. y) (λz. z)	→ λz. z
→ λz. z	

Term may have both terminating and non-terminating reduction sequences

(λu. λv. v) ((λx. x x)(λx. x x))
→ λv. v

(λu. λv. v) ((λx. x x)(λx. x x))
→ (λu. λv. v) ((λx. x x)(λx. x x))
→ ...

Reduction strategies – examples

Applicative-order may ***not*** be as efficient as normal-order when the argument is not used.

Normal-order	***Applicative-order***
(λx. p) ((λy. y) (λz. z))	(λx. p) ((λy. y) (λz. z))
→ p	→ (λx. p) (λz. z)
	→ p

Reduction strategies

- Similar to (but subtly different from) ***evaluation strategies*** in language theories
  - **Call-by-name** (like normal-order)
    - ALGOL 60
  - **Call-by-need** ("memorized version" of call-by-name)
    - Haskell, R, ...
  - **Call-by-value** (like applicative-order)
    - C, ...
  - ...

arguments are not evaluated, but directly substituted into function body

called "lazy evaluation"

called "eager evaluation"

Programming in λ-calculus

- Encoding Boolean values and operators
  - True ≡ λx. λy. x
  - False ≡ λx. λy. y
  - not ≡ λb. b False True

not True
→ True False True
→ False

not False
→ False False True
→ True

Main points till now

- Syntax: notation for defining functions
  - "Pure": without adding any additional syntax

    (Terms) M, N ::= x | λx. M | M N
- Semantics (reduction rules)

  $(\lambda x.\ M)\ N \rightarrow [N/x]M \quad (\beta)$
- Next: programming in "pure" λ-calculus
  - Encoding data and operators

Programming in λ-calculus

- Encoding Boolean values and operators
  - True ≡ λx. λy. x
  - False ≡ λx. λy. y
  - not ≡ λb. b False True
  - and ≡ λb. λb'. b b' False

and True b
→* True b False
→ b

and False b
→* False b False
→ False

Programming in λ-calculus

- Encoding Boolean values and operators
  - True ≡ λx. λy. x
  - False ≡ λx. λy. y

Programming in λ-calculus

- Encoding Boolean values and operators
  - True ≡ λx. λy. x
  - False ≡ λx. λy. y
  - not ≡ λb. b False True
  - and ≡ λb. λb'. b b' False
  - or ≡ λb. λb'. b True b'

or True b
→* True True b
→ True

or False b
→* False True b
→ b

Programming in λ-calculus

- Encoding Boolean values and operators
  - True ≡ λx. λy. x
  - False ≡ λx. λy. y
  - not ≡ λb. b False True
  - and ≡ λb. λb'. b b' False
  - or ≡ λb. λb'. b True b'
  - if b then M else N ≡ b M N

Not unique encoding

Programming in λ-calculus

- Church numerals
  - $\underline{0}$ ≡ λf. λx. x *(the same as False!)*
  - $\underline{1}$ ≡ λf. λx. f x
  - $\underline{2}$ ≡ λf. λx. f (f x)
  - $\underline{n}$ ≡ λf. λx. f^n x
  - succ ≡ λn. λf. λx. f (n f x)

succ $\underline{n}$
→ λf. λx. f ($\underline{n}$ f x)
= λf. λx. f ((λf. λx. f^n x) f x)
→ λf. λx. f (f^n x)
= λf. λx. f^{n+1} x
= $\underline{n+1}$

Programming in λ-calculus

- Encoding Boolean values and operators
  - True ≡ λx. λy. x
  - False ≡ λx. λy. y
  - not ≡ λb. b False True
  - and ≡ λb. λb'. b b' False
  - or ≡ λb. λb'. b True b'
  - if b then M else N ≡ b M N
  - not' ≡ λb. λx. λy. b y x

not' True
→ λx. λy. True y x
→ λx. λy. y = False

not' False
→ λx. λy. False y x
→ λx. λy. x = True

Programming in λ-calculus

- Church numerals
  - $\underline{0}$ ≡ λf. λx. x
  - $\underline{1}$ ≡ λf. λx. f x
  - $\underline{2}$ ≡ λf. λx. f (f x)
  - $\underline{n}$ ≡ λf. λx. f^n x
  - succ ≡ λn. λf. λx. f (n f x)
  - succ' ≡ λn. λf. λx. n f (f x)

Programming in λ-calculus

- Church numerals
  - $\underline{0}$ ≡ λf. λx. x *(the same as False!)*
  - $\underline{1}$ ≡ λf. λx. f x
  - $\underline{2}$ ≡ λf. λx. f (f x)
  - $\underline{n}$ ≡ λf. λx. f^n x

Programming in λ-calculus

- Church numerals
  - $\underline{0}$ ≡ λf. λx. x
  - $\underline{1}$ ≡ λf. λx. f x
  - $\underline{2}$ ≡ λf. λx. f (f x)
  - $\underline{n}$ ≡ λf. λx. f^n x
  - succ ≡ λn. λf. λx. f (n f x)
  - iszero ≡ λn. λx. λy. n (λz. y) x

iszero $\underline{0}$
→ λx. λy. $\underline{0}$ (λz. y) x
= λx. λy. (λf. λx. x) (λz. y) x
→ λx. λy. (λx. x) x
→ λx. λy. x = True

iszero $\underline{1}$
→ λx. λy. $\underline{1}$ (λz. y) x
= λx. λy. (λf. λx. f x) (λz. y) x
→ λx. λy. (λx. (λz. y) x) x
→ λx. λy. ((λz. y) x)
→ λx. λy. y = False

iszero (succ $\underline{n}$) →* False

Programming in λ-calculus

- Church numerals
  - $\underline{0} \equiv \lambda f.\, \lambda x.\, x$
  - $\underline{1} \equiv \lambda f.\, \lambda x.\, f\ x$
  - $\underline{2} \equiv \lambda f.\, \lambda x.\, f\ (f\ x)$
  - $\underline{n} \equiv \lambda f.\, \lambda x.\, f^n\ x$
  - succ $\equiv \lambda n.\, \lambda f.\, \lambda x.\, f\ (n\ f\ x)$
  - iszero $\equiv \lambda n.\, \lambda x.\, \lambda y.\, n\ (\lambda z.\, y)\ x$
  - add $\equiv \lambda n.\, \lambda m.\, \lambda f.\, \lambda x.\, n\ f\ (m\ f\ x)$
  - mult $\equiv \lambda n.\, \lambda m.\, \lambda f.\, n\ m\ f$

Programming in λ-calculus

- Booleans
- Natural numbers
- Pairs
- Lists
- Trees
- Recursive functions
- …

Read supplementary materials on course website